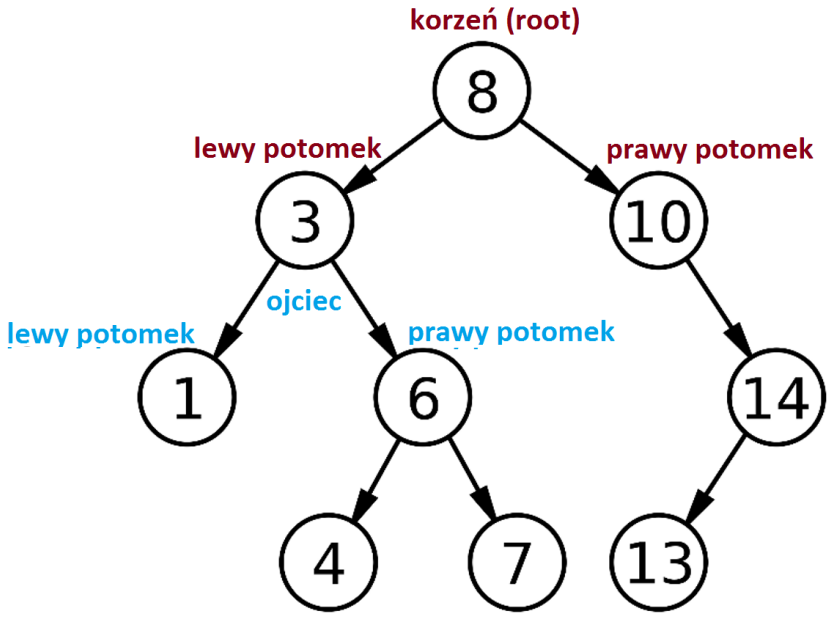
Binarne drzewo poszukiwań - *Binary Search Tree (BST)*

Drzewo to hierarchiczna struktura danych. Co to znaczy? Że do jego „obsługi” w kodzie będziemy musieli używać rekurencji (tej trudnej i nieciekawej). Na początku przyjrzymy się, jak to wszystko wygląda na obrazkach, dopiero potem zaimplementujemy naszą wiedzę w kodzie.

Drzewo składa się z węzłów (**nodes**). Każdy z nich posiada co najwyżej dwóch *następników.* Stąd też nazwa „binarne”, bo binarny to „dwójkowy”, zawierający dwa elementy). Drzewo posiada tzw. „węzeł nadrzędny” (**root**). Jego następniki są nazywane węzłami *potomnymi (dziecko, potomek) (**child nodes**).*



Istnieje jedna i podstawowa reguła drzewa binarnego – Wszystkie elementy znajdujące się w lewym poddrzewie są mniejsze od swojego ojca, natomiast elementy w prawym poddrzewie są większe od swojego ojca. Reguła to obowiązuje zawsze i wszędzie, na wszystkie poddrzewa.

A co z elementami równymi? To już kwestia własnego ustalenia. Na zajęciach takie elementy wrzucaliśmy na prawo.

Na obrazku rootem (głównym węzłem) drzewa jest liczba 8. Ma ona dwóch potomków: 3 oraz 10. Następnie 3 też ma dwóch potomków: 1 oraz 6. Można więc powiedzieć, że 3 jest ojcem dla 1 i 6. Bądź 14 jest ojcem 13; albo że 8 jest ojcem 3 i 10. Warto zauważyć, że np. taka 14 posiada tylko lewego potomka 13. Węzeł 4 nie ma potomków w ogóle.

Na tym zakończylibyśmy tą całą otoczkę teoretyczną. Nie omówiliśmy takich rzeczy jak dodawanie węzła, przeszukiwanie drzewa, itp. To wszystko można znaleźć w tym filmiku: **https://youtu.be/_V7a1Gwuj5k?t=37m46s** (od **37:46** do **45:20**). Gościu fajnie tłumaczy, ALE nie polecam sugerować się jego kodem. BO jego drzewo jest w C++ ORAZ po drugie i najważniejsze – stosuje **tablicową implementację drzewa**, która do niczego nam się nie przyda. Bo my musimy napisać drzewo „z prawdziwego zdarzenia”, a nie jakieś tablicowe śmieszki.

Warto jeszcze nadmienić, że w tym pdfie nie będzie pokazane całe drzewo ze wszystkimi jego możliwościami, itd. Po taką wiedzę zapraszam piętro niżej, do wydziału leśnego. No bo kto jak kto, ale oni o drzewach wiedzą najwięcej.

Implementacja

Tworzymy klasę „węzła” (**node).**

```
public class Node
{
    int number;
    Node left;
    Node right;
}
```

Klasa nazywa się Node, czyli węzeł. Zawiera trzy pola: **number**, **left** oraz **right**.

- `int number;` - to liczba, czyli to, co dany węzeł ma przechowywać. W węzłach nie będziemy przetrzymywać wymyślnych klientów, adresów, czy innego barachła. Tylko zwykłą i najprostszą liczbę, tak jak to było wyżej na rysunku.
- `Node left;` - **referencja (odwołanie)** do lewego dziecka.
- `Node right;` - **referencja (odwołanie)** do prawego dziecka.

Na początku może to wyglądać dziwnie – klasa Node posiada dwa pola typu Node? Tak. Tak jak już było wspomniane – drzewo to struktura hierarchiczna, więc takie coś jest na początku dziennym. I takie coś trzeba będzie wertować rekurencyjnie (niestety).

Druga klasa, którą stworzymy zaraz pod kodem klasy węzła, będzie klasą drzewa:

```
public class Tree
{
    Node root;
    int counter;
}
```

Drzewo – czyli **Tree**. Klasa zawiera dwa pola:

- `Node root;` - **refrencja (odwołanie)** do głównego węzła, do *korzenia* tego drzewa
- `int counter;` - a tutaj taki mały licznik elementów drzewa, nie musi on tutaj być, ale dlaczego by go nie zrobić dla picu? ;)

Teraz zajmiemy się implementacją węzła (node)

```
public class Node
{
    int number;
    Node left;
    Node right;

    public Node(int value)    // konstruktor
    {
        this.number = value;   //1
        this.left = null;      //2
        this.right = null;     //2
    }

    public bool IsLeaf()
    {
        return (this.left == null && this.right == null);
    }
```

Dodaliśmy do kodu dwie funkcje. Pierwsza z nich to konstruktor, do którego przekazujemy tylko jeden parametr – wartość węzła, którą zaraz przypisujemy do pola **int numer (//1)**. W kolejnych dwóch linijkach przypisujemy odwołania do lewego i prawego dziecka tego węzła -> z racji tego, że ten węzeł jest „nowo narodzony” to nie ma dzieci, po prostu przypisujemy im nulle **(//2).**

Druga funkcja **bool IsLeaf()**, sprawdza, czy węzeł jest liściem. A węzeł jest liściem, gdy nie posiada dzieci. Każdy „nowo upieczony” węzeł będzie liściem, bo nie będzie posiadał ani lewego dziecka, ani prawego. Wtedy funkcja zwraca **true**. Gdy węzeł nie jest liściem, czyli posiada jedno lub dwoje dzieci, to funkcja zwraca **false.**

Kolejną funkcją będzie funkcja przeszukująca pod-węzły pod kątem danej wartości:

```
                // .............  //
        public Node Search(int value)  //0
        {
            if (this.number == value)      //1
            {
                return this;
            }
            else if (value < this.number)    //2
            {
                if (this.left == null)  //3
                {
                    return null;
                }
                else
                {
                    return this.left.Search(value);  //4
                }

            }
            else if (value > this.number)  //5
            {
                if (this.right == null)   //6
                {
                    return null;
                }
                else
                {
                    return this.right.Search(value);   //7
                }
            }
            return null;  //8
        }
```

Do funkcji **Node Search(int value)** wrzucamy wartość jaką chcemy znaleźć. Funkcja przeszukuje dany węzeł, dzieci tego węzła oraz dzieci-dzieci i dzieci-dzieci-dzieci, itd.

Gdy znajdzie – zwraca ten węzeł. Gdy nie znajdzie – zwraca null;

Przeanalizujmy tą funkcję. Składa się ona z trzech **ifów.** Pierwszy sprawdza, czy liczba której szukamy nie jest liczbą w obecnym węźle **(//1).** Jeśli tak, to od razu zwracamy ten węzeł i kończymy funkcję.

Drugi i trzeci wypadek jest bardziej złożony. Oba są na szczęście prawie że identyczne.

(//2) Jeśli szukana liczba jest mniejsza od tej z danego węzła, to wiadomo, że musimy szukać w lewym potomku tego węzła. Heh, ale jeśli nasz węzeł nie posiada lewego potomka to musimy zakończyć nasze poszukiwania fiaskiem **(//3).** Ale gdy istnieje lewy potomek to co? **(//4)** To wtedy szukamy w nim tej wartości. I teraz **uwaga!** Tutaj mamy tą *zdradziecką* rekurencję. Przeanalizujmy dokładnie tą linijkę:

```
return this.left.Search(value);
```

this.left to odwołanie do lewego dziecka naszego węzła. **this.left.Search()** to odwołanie się do funkcji **Search(),** czyli do tej w której obecnie jesteśmy, ale z tą różnicą, że wywołujemy ją dla lewego dziecka. Przekazujemy do niej to samo **value**, które widnieje tam wyżej **(//0).** Na samym początku linijki daliśmy **return**, przez co wszystko będzie wykonywać się rekurencyjne aż do znalezienia odpowiedniego węzła i jego zwrócenia **(//1).** Albo nieznalezienia i zwrócenia null **(//8)**.

Analogicznie postępujemy w trzeciej części tej funkcji, gdy szukana liczba jest większa lub równa od liczby przechowywanej w naszym węźle. **(//5).** Najpierw sprawdzamy, czy w ogóle istnieje prawy pod-węzeł **(//6).** Jeśli istnieje **(//7)** to zaczynamy przeszukiwać prawy węzeł i ewentualnie jego podwęzły (tak jak to miało miejsce z lewej strony).

Gdy żaden z tych 3 **ifów** się nie wykona, to pozostaje nam zwrócić null **(//8),** czyli informację o nieznalezieniu węzła o takiej wartości.

Dodawanie węzła do struktury

```
        // .............  //
        public void Add(int value)        //0
        {
            if (value >= this.number)    //1
            {
                if (this.right == null)
                {
                    this.right = new Node(value);  //2
                }
                else
                {
                    this.right.Add(value);    //3
                }
            }
            else if (value < this.number)    //4
            {
                if (this.left == null)
                {
                    this.left = new Node(value);    //5
                }
                else
                {
                    this.left.Add(value);     //6
                }
            }
        }
```

Funkcja **Add()** jest bliźniaczo podobna do omówionej wcześniej funkcji **Search()**. Przekazujemy do niej wartość, którą chcemy dodać do węzła (a dokładniej – do dzieci tego węzła)**(//0).** Funkcja składa się z dwóch głównych ifów **(//1),(//4)**. Pierwszy z nich **(//1)** wykona się wtedy, gdy liczba którą chcemy wpisać będzie większa od tej w aktualnym węźle. Jeśli tak się stanie, to najpierw sprawdzamy, czy dany węzeł posiada prawego potomka. Jeśli nie, to sprawa jest prosta. Tworzymy tego prawego potomka i przypisujemy mu naszą wartość **(//2).** W przeciwnym wypadku, gdy prawy węzeł istnieje, dzieje się rekurencja **(//3),** czyli znowu wywołujemy funkcję **Add()**, tylko z tą różnicą, że nie dla naszego węzła, a dla jego prawego potomka.

Dodawanie lewego węzła dzieje się analogicznie. W przypadku, gdy dodawana wartość jest mniejsza od tej w aktualnym węźle, to musimy wrzucić ją na lewo **(//4).** Gdy lewy pod-węzeł nie istnieje to sprawa jest prosta – tworzymy go z wartością którą chcemy dodać **(//5).** Gdy takowy węzeł istenieje, wywołujemy na nim funkcję **Add() (//6)**, identycznie jak wcześniej.

Wyświetlanie węzła i jego dzieci (i dzieci jego dzieci, itd.)

```
        public void Display()
        {
            if (this.left != null)
            {
                this.left.Display();     //1
            }
            Console.Write(" " + this.number);    //2
            if (this.right != null)
            {
                this.right.Display();     //3
            }
        }
```

}

Kolejną funkcją jest **Display()**, która będzie wyświetlała nasz węzeł, oraz wszystkie jego dzieci (i dzieci ich dzieci, itd.). Funkcja ta (jak i każda inna wcześniej przedstawiona) jest niestety rekurencyjna. Wyświetlenie węzła to po prostu wyświetlenie jego wartości, czyli zmiennej **number**. Wystarczy zwykłe **Console.Write()** ze spacją z przodu (albo z tyłu, kto jak tam woli).**(//2).** Ale wyświetlenie jednej wartości nie wystarczy – musimy przecież wyświetlić całą strukturę. Stąd też powyżej linijki **//2** oraz poniżej tworzymy dwa warunki. Ten powyżej **(//1)** sprawdza, czy nasz węzeł posiada lewe dziecko. Jeśli tak, to wywołuje na jego rzecz tą samą funkcję **Display().** Analogicznie dzieje się poniżej **(//3).** W przypadku gdy istnieje prawe dziecko to wywołujemy na jego rzecz funkcję **Display().**

Wreszcie możemy przejść do drzewa (tree)

```
public class Tree
{
    Node root;
    int counter;

    public Tree()       //1
    {
        root = null;
        counter = 0;
    }

    public bool IsEmpty()    //2
    {
        return this.root == null;
    }

    public void Add(int value)   //3
    {
        if (IsEmpty())
        {
            this.root = new Node(value);   //4
        }
        else
        {
            this.root.Add(value);     //5
        }
        counter++;
    }

    public bool Search(int value)     //6
    {
        if (this.root.Search(value) != null) return true;    //7
        else return false;
    }


    public void Display()    //8
    {
        if (IsEmpty() == false)   //9
        {
            this.root.Display();
        }
    }

    public int Count()   //10
    {
        return this.counter;
    }

}
```

Na początku tworzymy standardowy konstruktor. **(//1).** Do głównego korzenia drzewa (**root**) przypisujemy null (bo nowe drzewo jest zawsze puste), i licznik (**counter**) ustawiamy na **0**.

Kolejna funkcja to **IsEmpty() (//2).** Działanie jej jest proste – zwraca true gdy drzewo (czyli korzeń) jest pusty (**null**). W przeciwnym wypadku zwraca false.

Dalej mamy funkcję **Add()**, **(//3)** która dodaje element do drzewa. Jest ona bardzo prosta, a to dlatego, że całe właściwe dodawanie węzła znajduje się we wcześniej omówionym kodzie – w klasie Node. Na początku sprawdzamy, czy nasze drzewo jest puste (wszystko dzięki wcześniej napisanej funkcji IsEmpty()). Jeśli tak, to nowy węzeł musimy wrzucić na miejsce roota **(//4).** W przeciwnym wypadku, czyli gdy drzewo nie jest puste, nową wartość musimy upakować gdzieś w strukturze. Jednak cały kod dodawania już napisaliśmy, więc wystarczy wywołaj funkcję **Add()** z klasy **Node** dla naszego korzenia (**root**) **(//5).** Na końcu zwiększamy licznik elementów w drzewie.

Przedostatnią funkcją jest **Display(),** który wyświetla nasze całe drzewo. **(//8)** Dzieje się to tylko w przypadku, gdy nie jest ono puste **(//9).** Całe wyświetlanie polega na wywołaniu funkcji **Display()** z klasy **Node** dla naszego głównego węzła (**root**).

Ostatnia funkcja to **Count() (//10),** która zwraca ilość elementów w drzewie, czyli wartość naszego licznika.

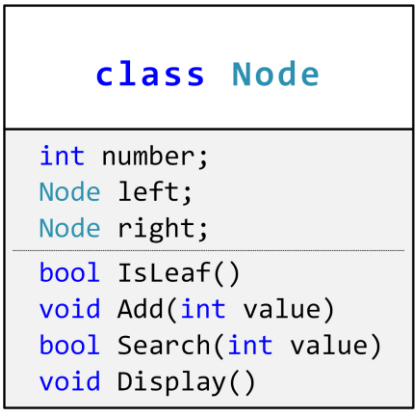
Cały kod drzewa i węzła znajduje się na końcu tego pliku.

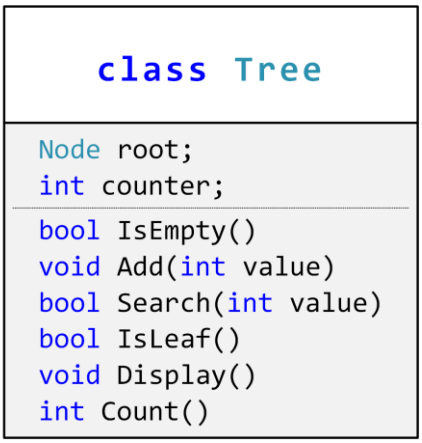
Wykorzystanie kodu drzewa w Main()

```
static void Main(string[] args)
{
      Tree brzoza = new Tree();  // stworzenie drzewa – brzozy
      brzoza.Add(7);    // bo sosna to tylko na opał się nadaje
      brzoza.Add(12);   // dodawanie elementów do drzewa
      brzoza.Add(4);
      brzoza.Add(1);
      brzoza.Add(8);
      brzoza.Add(12);
      brzoza.Add(63);
      brzoza.Add(2);

      Console.WriteLine("Drzewo posiada {0} elementów", brzoza.Count());

      brzoza.Display();  // wyświetlenie całego drzewa
      Console.WriteLine();

      Console.WriteLine("Czy w drzewie znajduje się liczba 4?");
      Console.WriteLine(brzoza.Search(4));

      Console.WriteLine("Czy w drzewie znajduje się liczba 23?");
      Console.WriteLine(brzoza.Search(23));

      brzoza.Add(23);

      Console.WriteLine("Czy w drzewie znajduje się liczba 23?");
      Console.WriteLine(brzoza.Search(23));

      Console.ReadKey();
}
```

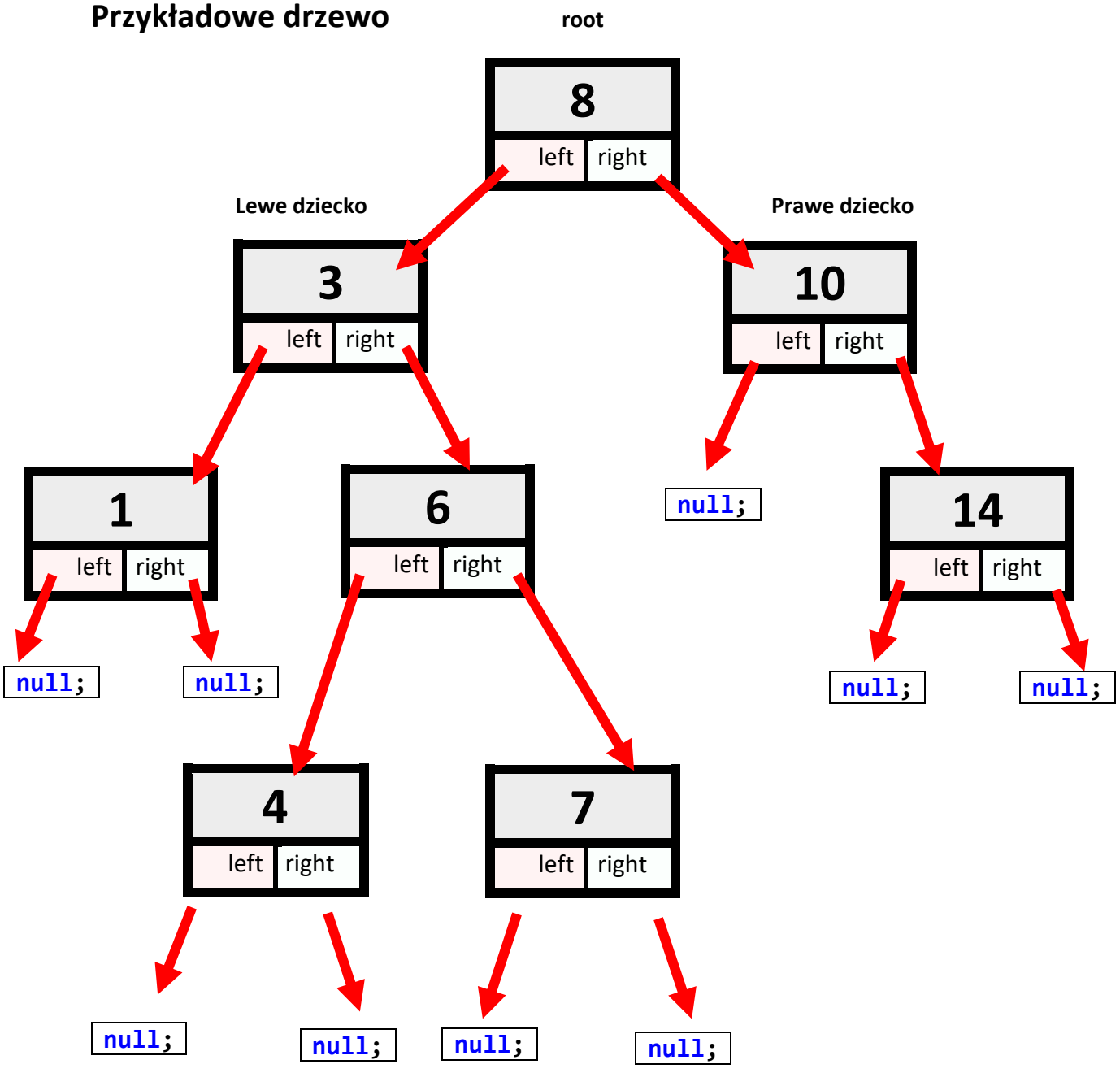
Kilka rysunków

Diagramy UML klas:





Przykładowe drzewo



Cały kod:

http://wklej.org/id/3021727/

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace tree
{
    public class Node
    {
        int number;
        Node left;
        Node right;

        public Node(int value)    // konstruktor
        {
            this.number = value;   //1
            this.left = null;      //2
            this.right = null;     //2
        }

        public bool IsLeaf()
        {
            return (this.left == null && this.right == null);
        }

        public Node Search(int value)  //0
        {
            if (this.number == value)      //1
            {
                return this;
            }
            else if (value < this.number)    //2
            {
                if (this.left == null)  //3
                {
                    return null;
                }
                else
                {
                    return this.left.Search(value);  //4
                }

            }
            else if (value > this.number)  //5
            {
                if (this.right == null)   //6
                {
                    return null;
                }
                else
                {
                    return this.right.Search(value);   //7
                }
            }
            return null;  //8
        }

        public void Add(int value)        //0
```

```
    {
        if (value >= this.number)    //1
        {
            if (this.right == null)
            {
                this.right = new Node(value);  //2
            }
            else
            {
                this.right.Add(value);    //3
            }
        }
        else if (value < this.number)    //4
        {
            if (this.left == null)
            {
                this.left = new Node(value);    //5
            }
            else
            {
                this.left.Add(value);     //6
            }
        }
    }

    public void Display()
    {
        if (this.left != null)
        {
            this.left.Display();     //1
        }
        Console.Write(" " + this.number);    //2
        if (this.right != null)
        {
            this.right.Display();     //3
        }
    }
}


public class Tree
{
    Node root;
    int counter;

    public Tree()       //1
    {
        root = null;
        counter = 0;
    }

    public bool IsEmpty()    //2
    {
        return this.root == null;
    }

    public void Add(int value)   //3
    {
        if (IsEmpty())
        {
            this.root = new Node(value);   //4
        }
        else
        {
            this.root.Add(value);     //5
        }
        counter++;
    }
```

```
        public bool Search(int value)     //6
        {
            if (this.root.Search(value) != null) return true;    //7
            else return false;
        }


        public void Display()    //8
        {
            if (IsEmpty() == false)   //9
            {
                this.root.Display();
            }
        }

        public int Count()   //10
        {
            return this.counter;
        }

    }

    class Program
    {
        static void Main(string[] args)
        {
            Tree brzoza = new Tree();  // stworzenie drzewa
            brzoza.Add(7);    // dodawanie elementów do drzewa
            brzoza.Add(12);
            brzoza.Add(4);
            brzoza.Add(1);
            brzoza.Add(8);
            brzoza.Add(12);
            brzoza.Add(63);
            brzoza.Add(2);

            Console.WriteLine("Drzewo posiada {0} elementów", brzoza.Count());

            brzoza.Display();  // wyświetlenie całego drzewa
            Console.WriteLine();

            Console.WriteLine("Czy w drzewie znajduje się liczba 4?");
            Console.WriteLine(brzoza.Search(4));

            Console.WriteLine("Czy w drzewie znajduje się liczba 23?");
            Console.WriteLine(brzoza.Search(23));

            brzoza.Add(23);

            Console.WriteLine("Czy w drzewie znajduje się liczba 23?");
            Console.WriteLine(brzoza.Search(23));

            Console.ReadKey();
        }
    }
}
```